Mechanized inductive proof of properties of a simple code optimizer

Alfons GESER

Universität Passau, Lehrstuhl für Programmiersysteme, D-94030 Passau
Phone: +49 851 509 353, E-mail: geser@fmi.uni-passau.de

Abstract. We demonstrate how mechanical proofs of properties of a simple code generator and a partial evaluator can be done by term rewriting induction. We yield proofs that the code generator is correct and that the partial evaluator produces equivalent, optimal, shorter code. We treat a case of disequations and show how comparisons can be done adequately.

1 Introduction

Although much effort has been devoted to automation of inductive reasoning, only a few trivial theorems can be proven fully automatically. It appears difficult enough to improve effectiveness and to increase the degree of automation of mechanical inductive proving for a strongly restricted domain of application. We restrict ourselves to claims and axioms that are universally quantified equations which can be directed so that they form a term rewriting system. We perform proofs by "implicit induction" [1]. We feed the prover a few lemmas, the way people use e.g. the Boyer/Moore theorem prover [4] very successfully.

The case study we are reporting, is a continuation of work that began in 1987. Rudolf Berghammer, Herbert Ehler, and Hans Zierer (BEZ, for short) gave an algebraic specification, using 59 term rewriting rules, of a code generator and partial evaluator for arithmetic expressions [3]. They proved correctness of the code generator and of the partial evaluator. They encoded parts of the proof such that they could employ RAP [11], a rapid prototyping tool for algebraic specifications. RAP uses a narrowing procedure to perform case analysis, and simplifies intermediate goals by rewriting. As it is not an inductive prover, inductive hypotheses had to be encoded as additional axioms, with the inductive variables as Skolem constants. As a further consequence, case analyses had to be provided explicitly by the user. In spite of these shortcomings, the case study was an obvious success as it illustrated a successful formal proof plan.

This encouraged Heinrich Hußmann, Ulrich Fraus, and the author to develop an inductive prover, TIP [7]. This tool essentially uses the data structures and algorithms RAP uses, and moreover manages inductive hypotheses. This paper is a summary of the author's experience proving BEZ's and two further claims using TIP. The extended system consists of 78 rewrite rules.

2 Related Work

Among the other studies which use algebraic methods to verify code generators, there are to mention: J. Strother Moore's KIT project, a very ambitious formal verification, using the Boyer/Moore prover, of a real life compiler for an abstract machine. William Young proved correctness of the code generator [19]. Compared to the KIT project, our approach is small scale. But it offers more automation as our prover needs no induction hints.

> "The key to the proof (*of the essential lemma for correctness of the code generator – A.G.*) is formulating the induction such that the inductive hypotheses fit together to yield a proof of the theorem for PROG2. Some measure of the complexity of the induction is that the induction hint, given in the form of a definition in the Boyer-Moore logic, has 12 parameters and is over 250 lines long."([19], p. 510)

The "algebraic alternative", surveyed in [16], uses homomorphism properties to guide the correctness proof. We do not follow this peculiar technique as we are interested in inductive proofs in a more general setting.

With the RAP and TIP tools, a number of other medium size case studies have been done, e.g. the formal specification of an industrial 8-bit microprocessor [9] (225 rewrite rules). Christian Rank formalized a code generator from a small functional language to a stack machine [14]. Heinrich Hußmann [13] analyzed the treatment of recursive function definitions, and attacked the problems of partiality and nontermination by a variant of fixed point induction.

3 A Short View to Theory

We assume that the reader is familiar with the essentials of algebraic specification and term rewriting. For surveys see [17] and [5], respectively. We will deal with hierarchical systems of simply typed first-order term rewriting systems.

A *term rewriting system* is a pair (Σ, R) where Σ is a signature (often omitted), and R is any (usually finite) binary relation on terms. The elements of R are called *rewrite rules*, and are written $l \to r$. The *rewrite relation*, $\to_R$, is defined as the smallest relation that contains R and is closed under instantiation by substitutions and under contexts. This mirrors the universal quantification of variables and the congruence property, respectively, of the described *semantic equality*, $\leftrightarrow_R^*$, the equivalence closure of $\to_R$. A term t from which some rewrite step $t \to_R u$ starts, is called *(R-)reducible*. In this case the subterm of t which is replaced is called the *redex*. If there is a derivation $s \to_R^* t$ where t is not reducible, then t is called a *normal form* of s.

A term rewriting system R is called *terminating* if no infinite derivation $t_1 \to_R t_2 \to_R \ldots$ exists. R is called *confluent* if for all terms s, t, $s \leftrightarrow_R^* t$ implies $s \to_R^* \leftarrow_R^* t$. That is, in a confluent term rewriting system, two terms are semantically equal only if they can be rewritten to a common descendant term. As it is well-known, for terminating, confluent term rewriting systems semantic

equality is decidable: $s \leftrightarrow_R^* t$ if and only if, $R(s) = R(t)$, where $R(s)$ denotes the unique normal form of s. Moreover, confluent rewriting systems guarantee for conservativity of extensions, and so (together with sufficient completeness, see below) for a clean hierarchy of specifications.

3.1 Inductive Theorems

Let us just recall briefly the theory of inductive proving in term rewriting. A term is called *ground* if it contains no variable. A term t is called *ground reducible* if every ground instance of t is reducible. Given a term rewriting system R, an equation $s \equiv t$ is called an *inductive theorem* if $s\sigma \leftrightarrow_R^* t\sigma$ holds for every substitution σ where both $s\sigma$ and $t\sigma$ are ground. A rewrite rule $s \to t$, which is an oriented equation, may likewise be called an inductive theorem. Let π be a position of a function symbol in l'. Then a *critical pair* of $l \to r$ below $l' \to r'$ at position π is, provided it exists, a pair of terms (c, p) if, roughly speaking, $c \leftarrow_{l \to r} t \to_{l' \to r'} p$ is most general (up to renaming of variables) among the forking derivations $\leftarrow_{l \to r} t' \to_{l' \to r'}$ where in t' the redex position of $\to_{l \to r}$ is at position π below the redex position of $\to_{l' \to r'}$. The term t in this derivation is called the *trigger* of the critical pair.

We employ a simplified version of Theorem 1 of Hofbauer/Kutsche to prove by implicit induction that a set H of claims are inductive theorems on a set R of axioms. The clue of the method is that both axioms and inductive claims are viewed as terminating term rewriting rules.

Theorem 1 [12]. *Let R and H be term rewriting systems such that*

1. *$R \cup H$ is terminating,*
2. *the left hand side of each rule in H is ground R-reducible,*
3. *every critical pair (c, p) of rules in R below rules in H satisfies*

$$c \to_{R \cup H}^* \leftarrow_{R \cup H}^* p \ .$$

Then H is a set of inductive theorems.

R may also include some previously proven lemmas. H is during the proof also used as the set of inductive hypotheses. Like in the Knuth/Bendix completion procedure, critical pairs which do not "join" are entered as new members into H.

Laurent Fribourg [8] has observed that condition (3) may be restricted to the set of critical pairs at position π, provided that π is *completely superposable*, i.e. the triggers formed by the critical pairs at π cover all ground instances of the left hand sides of H rules. If there is a completely superposable position then obviously (2) holds as well.

Computing the critical pairs of R rules below H rules is nothing but performing R-narrowing steps on H equations at the left hand side. In effect each serves to establish a *finite case analysis*. We will therefore call a position π in H where a critical pair exists, a *case analysis redex*. Instead of "π is completely

superposable" we will rather say that π offers a *complete case analysis.* If a case analysis is not complete, one of the missing cases may yield a counterexample.

In practice, function symbols are partitioned into constructors and evaluators. An evaluator symbol f is called *completely defined* if every term of the form $f(c_1, \ldots, c_n)$ where each c_i is a ground constructor term, is reducible. This property can be checked statically. If every evaluator is completely defined and no constructor term is reducible then *innermost* case analysis redexes are always completely superposable. Otherwise, we have to check on the spot that the respective case analysis is complete.

Typing takes care for the notion of a correctly typed term. One may commonly ignore typing information. However we wish to stress that ground reducibility and complete definedness mark an important exception. Obviously one would like to prove $x + y = y + x$ for all ground terms x, y of type `Nat`, but not for ground terms of other sorts, as e.g. `Stack`. In view of this, ground reducibility should not require $\texttt{empty} + \texttt{push}(0, \texttt{empty})$ to be reducible. For space reasons we do not develop theory for this question; we only take care that we get reducibility of *correctly typed* ground terms, which we feel should work. A point in favour of our conjecture is that complete definedness for correctly typed ground terms, together with termination, entails *sufficient completeness.*

3.2 Comparisons and Bi-rewriting

It is the common policy of the algebraic specification community to express every predicate other than equality by a Boolean valued function. While this encoding keeps the approach simple, it turns out very ineffective for the case of *transitive* binary relations. The basic idea behind the "bi-rewriting" approach of Jorge Levy and Jaume Agustí [15] is now to treat orders analogous to equality. Leo Bachmair and Harald Ganzinger extended it to the case of clausal reasoning [2].

An order $\leq$ is axiomatized by two sets, L and R, of term rewriting rules. The first, L, defines rewrite steps $s \to_L t$, such that $s \leq t$ holds. R, dually, defines rewrite steps $s \to_R t$ such that $s \geq t$ holds. Here we abuse notation: There may be function symbols in the signature which are interpreted as non-monotonic functions. Hence $\to_L$ and $\to_R$ need not be closed under contexts, and so are no proper rewrite relations. We define $\to_L$ and $\to_R$ to be the closure of L and R, respectively, under substitution (only). Rewrite steps thus may only be applied at the top of a term. To take into account equality, a congruence, we consider a third rewrite system, S, where $\to_S$ denotes the closure under contexts and substitution of S, as usual. So $\leq =_{\text{def}} (\to_L \cup \leftarrow_R \cup \leftrightarrow_S)^*$.

In practice a formal comparison of two terms, s and t, proceeds as follows. Term s is rewritten using L and S rules, and t is rewritten using R and S rules, to a common term. The name L is chosen to indicate that L rules may be applied only at the left hand side of a goal $s \leq t$. This rewriting process terminates if $\to_L \cup \to_R \cup \to_S$ is wellfounded.

This leads to a straightforward extension of theorem 1 towards comparisons. To this end, let the set of axioms and the set of claims each be partitioned into three subsets, indexed by L, R, and S, respectively.

Theorem 2 Bi-rewriting induction. *Let A_L, A_R, A_S, H_L, H_R, and H_S be term rewriting systems, and let $L =_{\text{def}} A_L \cup H_L$, $R =_{\text{def}} A_R \cup H_R$, $S =_{\text{def}} A_S \cup H_S$. Suppose that*

1. *$\to_L \cup \to_R \cup \to_S$ is wellfounded,*
2. *the left hand side of each rule in $H_L \cup H_R \cup H_S$ is ground A_S-reducible,*
3. *every critical pair (c, p) of rules in A_S below rules in H_S satisfies*

$$c \to_S^* \leftarrow_S^* p \ ,$$

4. *every critical pair (c, p) of rules in A_S below rules in H_L, and every critical pair (p, c) of rules in A_S below rules in H_R satisfies*

$$c\,(\to_S \cup \to_L)^*(\leftarrow_S \cup \leftarrow_R)^*\,p \ ,$$

Then H_S is a set of inductive theorems, and $s\sigma\ (\leftrightarrow_S \cup \to_L \cup \leftarrow_R)^*\ t\sigma$ holds for all $s \to t$ in L and all $t \to s$ in R, and for all substitutions σ where both $s\sigma$ and $t\sigma$ are ground.

We omit the proof.

4 The Specification

Now let us speak shortly about the specification of the code generator and the partial evaluator. We follow closely the specification given by BEZ [3]. Likewise, we use the input language of the specification tools RAP and TIP [6]. For the complete specification text cf. the workshop version [10]. An ASCII file is available, too; see Section 5 for details.

4.1 The Compiler

Assume given a small programming language for arithmetic expressions, by the following context free grammar.

Op ::= "+" | "−" | "∗" Expr ::= `Nat` | `Id` | Expr Op Expr

Here `Nat` and `Id` denote the set of natural numbers and of identifiers, respectively. With specifications given for numbers (`NAT`) and identifiers (`ID`), this grammar is easily translated into the a specification module `EXPRESSION` of arithmetic expressions.

Next one specifies an abstract data type module `ENVIRONMENT` for environments, i.e. finite mappings from identifiers to numbers. This gives one the means to speak about *source semantics*, specified formally in the module `SSEMANTICS`. The semantic mapping is specified as a function `func ssem: (Env,Expr)Nat`, by induction on the structure of arithmetic expressions. We take for granted that `NAT` contains definitions for the standard operations + (`add`), − (`sub`), and ∗ (`mult`).

Module INSTRUCTION enumerates the set of instructions to the stack machine. There are instructions to push a number (NSTORE(n)) or to push the value of an identifier (ISTORE(i)), and one per arithmetic operator (ADD, SUB, MUL), where e.g. ADD replaces the two topmost values on the stack by their sum value.

Stack machine programs, i.e. sequences of instructions, are modelled in module SEQUINSTRUCTION. The module defines constructor functions empty for the empty sequence, prefix for addition of an element to the left, and some evaluator functions, e.g. conc for concatenation. To BEZ's version, we add a function length, to be able to express length decrease in THM4 below, and functions bottom, upper, to obtain the last element of a sequence, and the rest of the sequence, respectively. The latter will be useful at the specification of the partial evaluator.

Then the target semantics TSEMANTICS, i.e. the stack machine interpreter, is modelled, based on a straightforward specification of stacks of natural numbers. The target semantics is given by a function tsem: (Env, SequInstr) Nat, specified using an auxiliary function ts: (Env,SequInstr,Stack)Stack by structural induction on the syntax of the target program.

The compiler module COMPILER introduces a function symbol compile: (Expr) SequInstr by induction on the structure of expressions.

Correctness of the compiler means that under any environment e, the target semantics applied to the compiled source program a yields the same value as the source semantics does. Our correctness claim reads as follows.[1]

(THM1) $\qquad \mathtt{tsem(e, compile(a))} = \mathtt{ssem(e, a)}$.

In subsection 5.1 we report on our TIP proof session for THM1.

4.2 The Partial Evaluator

There is a straightforward idea to partially evaluate target programs. Every pattern of the form

```
prefix(NSTORE(m), prefix(NSTORE(n), prefix(ADD, s)))
```

may be rewritten to the shorter form prefix(add(m, n), s), and likewise for SUB and MUL[2]. Let pev: (SequInstr) SequInstr be the function that iteratively replaces every occurrence of a pattern by its *contractum*. As BEZ mention, there is an obvious first order specification for pev, viz.

(1)

$$\forall\, \mathtt{l, s : SequInstr, m, n : Nat.}$$
$$\mathtt{pev(conc(l, prefix(NSTORE(m), prefix(NSTORE(n), prefix(ADD, s))))) =}$$
$$\mathtt{pev(conc(l, prefix(NSTORE(add(m, n)), s)))}$$

[1] In order to save space, and to drop information which is redundant for the reader, we will not use full TIP syntax, but a more compact representation.

[2] In their final version, BEZ dropped optimization of SUB and MUL patterns. We stay with their report version.

for ADD, and likewise for SUB and MUL, together with

$$(2) \qquad \forall\, t : \mathtt{SequInstr}. \quad \mathtt{OPTIMAL}(t) \implies \mathtt{pev}(t) = t$$

where OPTIMAL is a predicate on SequInstr defined by

$$\begin{aligned}
&\mathtt{OPTIMAL}(t) \iff \\
&\quad (\neg\exists\, l, s : \mathtt{SequInstr}, m, n : \mathtt{Nat} \\
&\qquad t = \mathtt{conc}(l, \mathtt{prefix}(\mathtt{NSTORE}(m), \mathtt{prefix}(\mathtt{NSTORE}(n), \mathtt{prefix}(\mathtt{ADD}, s)))) \vee \\
&\qquad t = \mathtt{conc}(l, \mathtt{prefix}(\mathtt{NSTORE}(m), \mathtt{prefix}(\mathtt{NSTORE}(n), \mathtt{prefix}(\mathtt{SUB}, s)))) \vee \\
&\qquad t = \mathtt{conc}(l, \mathtt{prefix}(\mathtt{NSTORE}(m), \mathtt{prefix}(\mathtt{NSTORE}(n), \mathtt{prefix}(\mathtt{MUL}, s)))))
\end{aligned}$$

Remark. Indeed pev is specified *uniquely* this way. This is proven by induction on the length of t. As a lemma, one needs to prove that contraction of an optimizable pattern does not destroy any other optimizable pattern.

In spite of the problem of expressiveness, we wish to stay reasoning within the framework of equations and rewriting. Our plan is as follows. We adopt BEZ's "algorithmic specification" of pev, given as a term rewriting system in module PARTEVAL. Then we prove by that pev satisfies certain equational properties derived from (1) and (2).

pev is based on an auxiliary function, pv. Given a split (l, s) of the target program, pv scans for an occurrence of a pattern at the start of the right half, s. One may distinguish whether the sequence s is too short to have a pattern, or whether it begins with a proper part of the pattern but does not continue accordingly, or whether s indeed begins with the pattern. In the latter case, after contraction of the pattern, one must be aware of a new pattern that is formed using the rightmost instruction of the left half, l.

In [3] the latter axioms have a form that is not satisfactory.

```
pv(postfix(l,x),prefix(NSTORE(m),prefix(NSTORE(n),prefix(ADD,s))))
-> pv(l,prefix(x,prefix(NSTORE(add(m,n)),s)))
```

Their left hand sides violate the constructor discipline: postfix(l,x) should be a constructor term, but postfix is not a constructor. Even worse, confluence does not hold. To fix this, we add access functions bottom and upper to SEQUINSTRUCTION. Then we replace postfix(l,x) by prefix(x',l') on the left hand side, and l by upper(prefix(l',x')) and x by bottom(prefix(l',x')), respectively, on the right hand side. Thus we get:

```
pv(prefix(x',l'),prefix(NSTORE(m),prefix(NSTORE(n),prefix(ADD,s))))
-> pv(upper(prefix(x',l')),
      prefix(bottom(prefix(x',l')),prefix(NSTORE(add(m,n)),s)))
```

Interesting to note, the new rewrite system is still terminating, although a semantic path order is necessary to prove it.

A basic theorem says that the partial evaluator preserves the semantics of a target program. The proof session for this theorem is reported in subsection 5.2.

$$(\mathtt{THM2}) \qquad \mathtt{tsem}(e, \mathtt{pev}(s)) = \mathtt{tsem}(e, s)$$

4.3 More about the Partial Evaluator

If PARTEVAL satisfies (1) and (2), then it satisfies THM2 as an inductive consequence. But THM2 does not imply (1) and (2). It is therefore advisable to prove further properties to increase confidence. We will prove that pev yields optimal instruction sequences, and decreases the length of an instruction sequence.

First we introduce a specification OPTIMALITY for a Boolean valued function optimal: (SequInstr) Bool, whose purpose is to internalize the predicate OPTIMAL.

(3) $\forall t : \mathtt{SequInstr}. \quad \mathtt{optimal(t)} = \mathtt{true} \iff \mathtt{OPTIMAL(t)}$

To define optimal by a term rewriting system we borrow the recursive structure from function pv. We should be able to prove the following claim, a consequence of (1), (2), and (3). The proof session is reported in subsection 5.3.

(THM3) $\mathtt{optimal(pev(s))} = \mathtt{true}$

Still it may be the case that optimal is not the wanted optimality predicate — it might be constantly true, for instance. There is a way to achieve sure knowledge. Show that optimal(s) does not rewrite to true for any s that contains an optimizable pattern. We cannot prove a negation by rewriting, but we can prove that optimal(s) yields false in that case. For the proof session see subsection 5.4.

(OPT)
$\mathtt{optimal(conc(t, prefix(NSTORE(m), prefix(NSTORE(n), prefix(ADD, s)))))}$
$= \mathtt{false}$

In order to derive a disequality, we employ confluence. Our rewriting system is confluent; and by confluence, a term cannot have two distinct normal forms, true and false. So we get

(4) $\forall\, \mathtt{t} : \mathtt{SequInstr}. \quad \mathtt{optimal(t)} = \mathtt{true} \implies \mathtt{OPTIMAL(t)}\ ,$

which is not yet the characterization (3) that we wanted but sufficient, together with theorem THM3, to imply

$$\forall\, \mathtt{t} : \mathtt{SequInstr}. \quad \mathtt{OPTIMAL(pev(t))}\ .$$

Even a function pev that satisfies both THM2 and THM3, may fail to satisfy (2), for it may change an optimal instruction sequence to another optimal one. Such a function may even increase the code size as follows.

```
pev(prefix(ISTORE(i),prefix(NSTORE(3),prefix(MUL,emptysequ)))) =
prefix(ISTORE(i),prefix(ISTORE(i),prefix(ISTORE(i),
                        prefix(ADD,prefix(ADD,emptysequ)))))
```

Regarding this it is interesting to learn that our partial evaluator does not increase code lengths. We report on our experience in subsection 5.5.

(THM4) $\mathtt{length(pev(s))} \leq \mathtt{length(s)}$

5 Proof Sessions

We used TIP with a peculiar choice of settings. Basically we orient every hypothesis and every lemma from left to right, just as the rewrite rules of the specification. We take care in each case that this orientation preserves termination of the induced rewriting relation. As a consequence, rewriting induction may correctly be applied, i.e. application of an inductive hypothesis needs not to be justified explicitly.

As steps are performed in a terminating way, we may drop some amount of user control. We have achieved good results switching the default "debug" mode off, but switching "interactive" mode on. In effect the user is only asked to select one of the offered case analysis redexes. Experience has shown that the success of a proof attempt strongly depends on the chosen redex position. Although the default *innermost* redex is most convenient, proof attempts using innermost redexes exclusively may fail; see lemma `CRIT` in section 5.2. Usually we succeeded when we chose the *outermost* redex provided that upon inspection the case analysis offered was complete.

The reader is encouraged to redo the sessions. The TIP system and the source text of this case study are available via anonymous FTP from server `forwiss.uni-passau.de` in directory `pub/local/tip`. Put in your home directory a copy of the settings file `tiprc.rewind` and rename it to `.tiprc`, to have TIP do rewriting induction by default. To start TIP, enter the command `tip comp.tip`. After the start, the input file `comp.tip` containing the specification text together with the inductive claims is read, and a check is made for termination and constructor totality.

Below we report how each proof is done in practice. We put stress on the way one is guided to the proof. For an overview of the dependence relation among the lemmas consult figure 1.

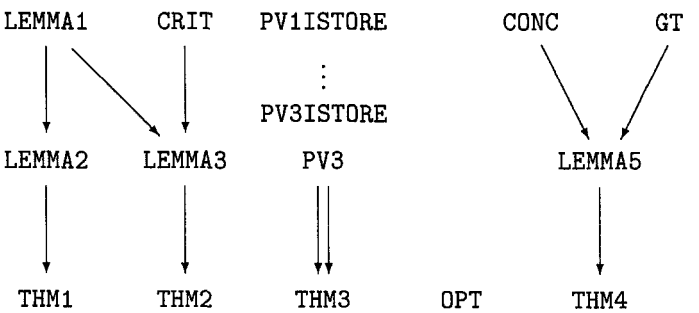

Fig. 1. The Proof Hierarchy

5.1 THM1

Having started TIP, we enter the proof session by typing `prove THM1`, then `go` to switch off the "debug" mode. The TIP system offers a case analysis redex,

```
? POSSIBLE INDUCTION REDEX:
top(ts(X1,compile(X2),emptystack)) = ssem(X1,X2) ,
```

which is acceptable[3]. The next offer shows that the proof is likely not to work.

```
? POSSIBLE INDUCTION REDEX:
top(ts(X1,conc(conc(compile(X5),compile(X6)),prefix(ADD,emptysequ)),
       emptystack))
= add(ssem(X1,X5),ssem(X1,X6))
```

We find ourselves faced with an equation where two `conc` symbols prevent the inductive hypothesis from application. As soon as we try to continue the proof attempt, even more `conc` symbols appear. We better find some lemma to get rid of the `conc` symbols. Now it is human intuition to find out that the execution of a concatenated sequence of instruction leads to the same state as subsequent execution of the parts.

(LEMMA1) $\qquad \mathtt{ts}(\mathtt{e}, \mathtt{conc}(\mathtt{s}, \mathtt{t}), \mathtt{k}) = \mathtt{ts}(\mathtt{e}, \mathtt{t}, \mathtt{ts}(\mathtt{e}, \mathtt{s}, \mathtt{k}))$

TIP can prove `LEMMA1` automatically. On demand it writes a proof documentation which is boring but fairly readable.

When we try to prove `THM1`, this time assuming `LEMMA1`[4], we are faced with a new problem.

```
? POSSIBLE INDUCTION REDEX:
add(top(pop(ts(X1,compile(X6),ts(X1,compile(X5),emptystack)))),
    top(ts(X1,compile(X6),ts(X1,compile(X5),emptystack))))
= add(ssem(X1,X5),ssem(X1,X6))
```

The inner `ts` term does not fit to the inductive hypothesis as the symbol `top` is missing. The outer `ts` term does not fit as its last argument is not `emptystack`. So let us generalize the claim in two ways: Drop the context, `top`, and replace the subterm, `emptystack`, by a variable. It leads to the following lemma.

(LEMMA2) $\qquad \mathtt{ts}(\mathtt{e}, \mathtt{compile}(\mathtt{a}), \mathtt{k}) = \mathtt{append}(\mathtt{ssem}(\mathtt{e}, \mathtt{a}), \mathtt{k})$

It is now straightforward to prove `THM1` by `LEMMA2`, and to prove `LEMMA2` by `LEMMA1`.

[3] TIP uses internal variable names of the form `X1`, `X2`, etc.

[4] In TIP, start a new session and type `enter lemma /assumed LEMMA1` .

5.2 THM2

Trying to prove claim THM2 without any lemma is soon recognized hopeless. The left hand side, l, of pv(l,s) must be generalized. The necessary lemma,

$$\mathtt{tsem}(\mathtt{e},\mathtt{pv}(\mathtt{l},\mathtt{s})) = \mathtt{tsem}(\mathtt{e},\mathtt{conc}(\mathtt{l},\mathtt{s}))\ ,$$

is slightly more special than BEZ's

(LEMMA3) $$\mathtt{ts}(\mathtt{e},\mathtt{pv}(\mathtt{l},\mathtt{s}),\mathtt{k}) = \mathtt{ts}(\mathtt{e},\mathtt{s},\mathtt{ts}(\mathtt{e},\mathtt{l},\mathtt{k}))$$

But the proofs are essentially the same. So we will stay with LEMMA3. A proof attempt for LEMMA3 unveils at once that LEMMA1 is needed. With LEMMA1 assumed, we arrive at

```
? POSSIBLE INDUCTION REDEX:
ts(X1,prefix(bottom(prefix(X52,X53)),prefix(NSTORE(add(X54,X55)),X56)),
      ts(X1,upper(prefix(X52,X53)),X4))
= ts(X1,X56,append(add(X54,X55),ts(X1,prefix(X52,X53),X4))) ,
```

which shows us that the prover cannot deal appropriately with the theory of bottom and upper. One would like to try with the equality

(CUB) $$\mathtt{conc}(\mathtt{upper}(\mathtt{prefix}(\mathtt{x},\mathtt{s})),\mathtt{prefix}(\mathtt{bottom}(\mathtt{prefix}(\mathtt{x},\mathtt{s})),\mathtt{t})) = \mathtt{prefix}(\mathtt{x},\mathtt{conc}(\mathtt{s},\mathtt{t}))\ ,$$

which expresses in a general way that conc is left-inverse to (upper, bottom), which split a nonempty sequence. Using CUB in the proof of LEMMA3, however, produces no recognizable effect. The reason is that the symbol conc of the left hand side is unlikely to appear thanks to the presence of LEMMA1 which we introduced for the purpose to get rid of conc symbols. To have CUB working, one has to take the effect of LEMMA1 into account, i.e. one has to compute the critical pair between LEMMA1 and CUB. This yields the following lemma.

(CRIT)
$$\mathtt{ts}(\mathtt{e},\mathtt{prefix}(\mathtt{bottom}(\mathtt{prefix}(\mathtt{x},\mathtt{l})),\mathtt{s}),\mathtt{ts}(\mathtt{e},\mathtt{upper}(\mathtt{prefix}(\mathtt{x},\mathtt{l})),\mathtt{k})) = \mathtt{ts}(\mathtt{e},\mathtt{s},\mathtt{ts}(\mathtt{e},\mathtt{prefix}(\mathtt{x},\mathtt{l}),\mathtt{k}))$$

There are two ways to prove CRIT, either as an immediate consequence of CUB and LEMMA1, or without CUB. We find the latter technically more suggestive. For the proof session, we only remark that the first offer for an inductive redex,

```
ts(X1,prefix(bottom(prefix(X8,X9)),X4),
      ts(X1,prefix(X2,upper(prefix(X8,X9))),X5))
= ts(X1,X4,ts(X1,prefix(X2,prefix(X8,X9)),X5)) ,
```

should be rejected. A good reason to do so is the fact that an occurrence of symbol prefix in the second line prevents the inductive hypothesis from being applied, a fact that is not changed when one accepts this offer. CRIT is a witness that the leftmost-innermost redex selection strategy may fail.

5.3 THM3

Unlike previous claims, THM3 is not straightforward to generalize by one proposition. For example, the conjecture that pv turns an optimal left argument into an optimal result, is wrong.

(THM3A) $\texttt{optimal}(\texttt{pv}(\texttt{l},\texttt{s})) = \texttt{optimal}(\texttt{l})$

The TIP prover, running in "debug mode", finds a counterexample for it. In a nutshell, the intermediate claim

```
optimal(conc(l,prefix(ADD,emptysequ))) = optimal(l)
```

turns out not to hold for the case

```
l = prefix(NSTORE(m),prefix(NSTORE(n),emptysequ))
```

The optimal instruction sequence l is supplemented to an instruction sequence which is not optimal, such that pv does not take action for it.

The suggestion we finally followed is guided by syntactic considerations. The trial to prove THM3 leads to an infinite sequence of claims whose left hand sides follow the pattern

```
optimal(pv(prefix(ISTORE(i),l),r) .
```

We know that any prefix of the form ISTORE(i) of the left argument of pv cannot be part of an optimizable pattern. In other words, ISTORE(i) may as well be stripped. Indeed it is routine to prove the following lemma.

(PV1ISTORE) $\texttt{optimal}(\texttt{pv}(\texttt{prefix}(\texttt{ISTORE}(\texttt{i}),\texttt{l}),\texttt{r})) = \texttt{optimal}(\texttt{pv}(\texttt{l},\texttt{r}))$

After trying with this lemma, we learn that another pattern occurs,

```
optimal(pv(prefix(ADD,l),r) .
```

Again we prove a lemma, PV1ADD, to express that the first instruction may be stripped.

Repeating this procedure, one can construct a finite set of lemmas PV1ISTORE, PV1ADD, PV1SUB, PV1MUL, PV2ISTORE, PV2ADD, PV2SUB, PV2MUL, PV3ISTORE, PV3 (three NSTORE instructions at the beginning). Each of these lemmas can be proven for itself, and together they allow to prove THM3. The lemmas mirror the case analysis structure of optimal.

5.4 OPT

The optimality proof is comparatively easy. Basically, one should prefer to choose the case analysis redex at the symbol optimal, provided that this yields a complete case analysis. This holds whenever at least three prefix symbols appear at the top of the argument of optimal.

5.5 THM4

Here the main problem is the adequate treatment of the $\leq$ relation on natural numbers. First let us demonstrate that the naive modelling by a Boolean valued function `le` is not satisfying.

Given a rewriting system for `le` on the naturals, let us attempt to prove the following claim.

(THM4A) $\qquad \texttt{le(length(pev(s)),length(s))} = \texttt{true}$

During a proof session for THM4A soon problems emerge like those experienced for THM3. It is natural to try the same methods to solve them. One may prove lemma LE1ISTORE for instance.

(LE1ISTORE)
$$\texttt{length(pv(prefix(ISTORE(i),l),s))} = \texttt{succ(length(pv(l,s)))}$$

There is, however, no such lemma for the case

```
length(pv(prefix(NSTORE(m),prefix(NSTORE(n),prefix(NSTORE(p),l)),s))
```

as one cannot predict uniformly how many of the NSTORE symbols pv will remove. This leaves to say only that the length decreases when the first NSTORE symbol is stripped from the instruction sequence. At the following claim, one may experience why any proof attempt, even supported by all lemmas LE1ISTORE, ..., LE3ISTORE, must fail.

(LEN) $\qquad \texttt{le(length(pv(emptysequ,prefix(NSTORE(m),s))),}$
$\texttt{succ(length(pv(emptysequ,s))))} = \texttt{true}$

The offer for a case analysis redex

```
? POSSIBLE INDUCTION REDEX:
le(length(pv(emptysequ,prefix(NSTORE(add(X1,X60)),X61))),
   succ(succ(succ(length(pv(emptysequ,X61))))))  = true
```

indicates that two succ symbols prevent TIP from applying the inductive hypothesis. Only an application of the transitivity law for `le` is necessary to close the gap. Transitivity of binary Boolean functions however is not supported by rewriting.[5]

As $\leq$ is a transitive relation, we may also try bi-rewriting induction (theorem 2). All rewrite rules treated so far concern the semantic equality, and so are collected in the set A_S. It is easy to see that the sets A_R and H_L contain the only rule `succ(x)` $\rightarrow$ `x` and `length(pev(s))` $\rightarrow$ `length(s)`, respectively. All other sets are empty.

To simulate bi-rewriting in TIP, we add a new rewrite rule for each comparison as if the comparison symbol were the equality symbol. To supervise that L and R rules and hypotheses are applied correctly, we introduce each rule as a lemma, and we switch TIP to

[5] It can be expressed as a *conditional* rewriting rule, but conditional rewriting is far less automatable at the moment.

```
set param hypothesis_usage interlr
set param lemma_usage interlr
```

having the effect that before every application of an inductive hypothesis or lemma, the user is asked whether the intended step should take place.

On these grounds, the axiom for ">" is encoded as an unproven lemma `GT`, `succ(x) = x`. Likewise the inductive claim `THM4` is encoded as `length(pev(s)) = length(s)`. With that we can prove `THM4` using the encoding of the following obvious `LEMMA5` (in L),

(`LEMMA5`) $\qquad \texttt{length}(\texttt{pv}(\texttt{l},\texttt{s})) \leq \texttt{length}(\texttt{conc}(\texttt{l},\texttt{s}))$.

In the same way, we achieved a proof of `LEMMA5` by `GT` where we use the equational lemma (in S)

(`CONC`) $\qquad \texttt{length}(\texttt{conc}(\texttt{s},\texttt{t})) = \texttt{add}(\texttt{length}(\texttt{s}),\texttt{length}(\texttt{t}))$.

6 Conclusion

We revisited Berghammer, Ehler, and Zierer's study [3] on automated inductive reasoning. Our inductive prover, TIP, set for term rewriting induction, yields complete proofs with a degree of automation of typically $> 98\%$, which is the number of internal steps divided by the total number of steps.

We extended the case study to show how a negative proposition, and how a comparison can be attacked by rewriting induction.

Acknowledgements. Thanks to Rudi Berghammer and Bettina Buth for giving me the opportunity to present my work in Kiel. I am grateful to Gerald Lüttgen for reading and commenting on a preliminary version.

References

1. Leo Bachmair. Proof by consistency in equational theories. In *3rd Proc. IEEE Symp. Logic in Computer Science*, pages 228–233, July 1988.
2. Leo Bachmair and Harald Ganzinger. Rewrite techniques for transitive relations. Technical Report MPI-I-93-249, Max-Planck-Institut für Informatik, Saarbrücken, Germany, November 1993.
3. Rudolf Berghammer, Herbert Ehler, and Hans Zierer. Towards an algebraic specification of code generation. *Science of Computer Programming*, 11:45–63, 1988. Also as technical report TUM-I8707, June, 1987, Technische Universität München, Germany.
4. Robert S. Boyer and J. Strother Moore. *A computational logic handbook.* Academic Press, 1988.
5. Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, chapter 6, pages 243–320. Elsevier, 1990.
6. Ulrich Fraus. Inductive theorem proving for algebraic specifications — TIP system user's manual. Technical Report MIP-9401, Universität Passau, Germany, February 1994.

7. Ulrich Fraus and Heinrich Hußmann. Term induction proofs by a generalization of narrowing. In C. Rattray and R. G. Clark, editors, *The Unified Computation Laboratory — Unifying Frameworks, Theories and Tools*, Oxford, UK, 1992. Clarendon Press.
8. Laurent Fribourg. A strong restriction of the inductive completion procedure. *J. Symbolic Computation*, 8(3):253–276, September 1989.
9. Alfons Geser. A specification of the intel 8085 microprocessor — a case study. In *[18]*, pages 347–402, 1987.
10. Alfons Geser. Mechanized inductive proof of properties of a simple code optimizer. In Bettina Buth and Rudolf Berghammer, editors, *Systems for Computer-Aided Specification, Development, and Verification.* Technical report 9416, Universität Kiel, Germany, October 1994.
11. Alfons Geser and Heinrich Hußmann. Experiences with the RAP system — a specification interpreter combining term rewriting and resolution. In Bernard Robinet and Reinhard Wilhelm, editors, *2nd European Symposium on Programming*, pages 339–350. Springer LNCS 213, March 1986.
12. Dieter Hofbauer and Ralf-Detlef Kutsche. Proving inductive theorems based on term rewriting systems. In *Proc. Algebraic and Logic Programing*, pages 180–190, Gaußig, Germany, 1988. Springer LNCS 343.
13. Heinrich Hußmann. A case study towards algebraic specification of code generation. In Maurice Nivat, C. Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Algebraic Methodology and Software Technology 91*, Workshops in Computing, pages 254–263. Springer, 1992.
14. Heinrich Hußmann and Christian Rank. Specification and prototyping of a compiler for a small applicative language. In *[18]*, pages 403–418, 1987.
15. J. Levy and J. Agustí. Bi-rewriting, a term rewriting technique for monotonic order relations. In Claude Kirchner, editor, *Int. Conf. Rewriting Techniques and Applications*, pages 17–31. Springer LNCS 690, 1993.
16. Teodor Rus. Algebraic alternative for compiler construction. In *IMA Conf. on the Unified Computation Laboratory*, pages 144–152, Stirling, Scotland, 1990.
17. Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science, Vol. B.* Elsevier - The MIT Press, 1990.
18. Martin Wirsing and Jan A. Bergstra. *Algebraic methods: Theory, Tools, and Applications.* Springer LNCS 394, June 1987.
19. William D. Young. A mechanically verified code generator. *J. Automated Reasoning*, 5(4):493–518, 1989.